IEEE Ninth International Conference on Computer and Information Technology

# Dynamic Adaptation of the Master-Worker Paradigm

Françoise André, Guillaume Gauvrit, Christian Pérez
IRISA / INRIA
Campus de Beaulieu, 35042 Rennes cedex, France
{Francoise.Andre, Guillaume.Gauvrit, Christian.Perez}@irisa.fr

***Abstract*—The size, heterogeneity and dynamism of the execution platforms of scientific applications, like computational grids, make using those platforms complex. Furthermore, today there is no effective and relatively simple solution to the programming of these applications independently of the target architectures. Using the *master-worker* paradigm in software components can provide a high level abstraction of those platforms, in order to ease their programming and make these applications portable. However, this does not take into account the dynamism of these platforms, such as changes in the number of processors available or the network load.**

**Therefore we propose to make the master-worker abstraction dynamically adaptable. More specifically, this paper characterizes the master-worker paradigm on distributed platforms, then describes a decision algorithm to switch between master-worker implementations at run-time.**

***Keywords-dynamic adaptation; master-worker paradigm; software engineering; grid computing;***

## I. Introduction

Distributed systems range from single computers with a multicore processor to grids that aggregates many computer clusters. They are usually composed of many computers of various kind that are connected by heterogeneous networks. Their size and heterogeneity makes them complex to program. Besides, the variability of resources and their sharing among users make these systems dynamic, especially computational grids. Indeed, new computers may be added to the grid while some may go to maintenance, and the workload as well as the network load can vary significantly during the execution of a task. Additionally, the applications developed for these systems are increasingly complex and thus hard to design. Today, there is no efficient and relatively simple solutions to program applications for distributed systems regardless of the target infrastructure.

These distributed systems are greatly used to run scientific or engineering applications that need a large amount of computing power. For instance in molecular biology to fold proteins, or in avionics to study air flow. Many of these applications are parametric ones, where different instances of the same code are executed in parallel on varying parameters. These parametric applications can be well designed by using the master-worker paradigm, hence the number of master-worker software or middleware one can encounter like SETI@Home [1], NetSolve [2] or DIET [3]. So, in this paper, we focus only on master-worker software aimed to be run on distributed systems. Some of these middleware are well suited for low scale applications while others are a better fit for high scale or highly dynamic infrastructures. Today, one has to choose at design time between all these alternatives to develop a master-worker application, and one cannot switch easily between them when developing, even less at run time.

So, to deal with these difficulties, in the aim to ease the development of parametric applications for distributed systems, we study in this paper how to dynamically adapt the master-worker paradigm, focusing on the creation of an algorithm to decide when to adapt. Section II presents various master-worker implementations. Section III describes a way to adapt the master-worker paradigm. Then Section IV characterizes master-worker applications on distributed systems to build foundations for an algorithm to make adaptation choices, which is described in Section V Finally, a conclusion ends this paper.

## II. Master-Worker implementations

The software or middleware implementing the master-worker paradigm is usually composed of five parts: the *master* sends tasks to compute to *workers*, *monitors* collect information about the state of the hardware and software resources which is stored in a *database*, and a *scheduler* selects to which workers the tasks are to be sent according to their kind and the informations stored in the database.

Many alternatives can be used to implement the master-worker paradigm, each better fitted to different situations.

The obvious one is to hard-code the paradigm in the application, with the scheduler and the database into the master. The worker would be chosen according to the *round-robin* pattern, that is one after the other, cycling. No monitor would be used. One drawback of this approach is the poor separation of concern; one other is that the dynamism of the grid or cluster is not at all taken into account. However, this implementation can be fast, simple to implement and well suited to software to be run on private clusters (not shared) or for prototypes.

Another alternative is to use a framework, like NetSolve [2]. It is composed by a *master*, an *agent* and *workers*, as shown on Figure 1, where each worker should be deployed on a different computer. Here, the *scheduler* and the *database* are in the agent, while the workers can *monitor* the speed and the load of their computer as well as the latency and bandwidth of the network between them and the agent. This information is sent

978-0-7695-3836-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CIT.2009.40



IEEE computer society

Authorized licensed use limited to: UR Rennes. Downloaded on December 14, 2009 at 04:42 from IEEE Xplore. Restrictions apply.

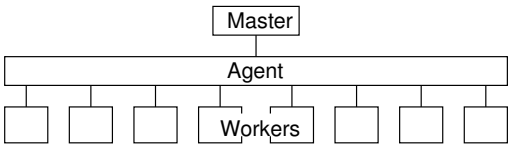


Fig. 1. NetSolve

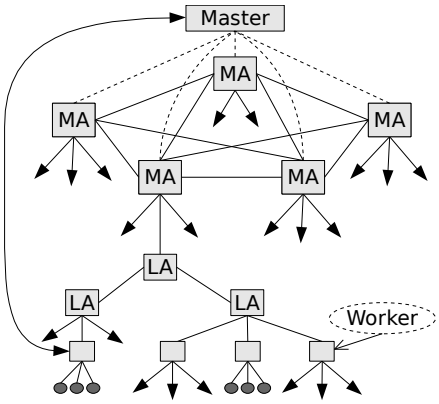


Fig. 2. DIET

to the agent just after the deployment of the workers. When the master has to send a request, it asks the agent for a list of workers able to compute the task, then the master tests the delay to the workers and sends the request to the fastest one.

This kind of centralized implementation of the paradigm solves the problem of the separation of concerns and it can in part deal with the dynamism of the resources. Nevertheless, this centralized architecture may not be well suited to large distributed systems, like large grids.

DIET [3] is specifically developed for those large distributed systems. As shown on Figure 2, DIET is composed by four elements: *masters*, *master agents* (MA), *local agents* (LA), and *workers*. The schedulers are only in the MAs, and the monitors at the level of the workers. The agents should be distributed according to the network topology, for example one MA by grid and one LA by cluster. The LAs are used to relay requests and information between MAs and workers. A LA stores various local information useful to distribute requests. However, the costs of this implementation can be significant for software using very short tasks.

NINF [4] and Nimrod/G [5] can be cited among other available frameworks.

The differences among these implementations makes them fit differently to various situations. For example, when an application with a dynamic behavior switches from short tasks of about 10 ms, to longer tasks averaging 100 s, it might be beneficial to switch from a simple and fast implementation like round-robin to another one better able to distribute the workload, like NetSolve or DIET. However, this is not currently feasible, unless if this logic is implemented in the application, which breaks the separation of concerns. A better solution would be to use an abstraction of these implementations of the paradigm and let it switch between different implementations when needed.

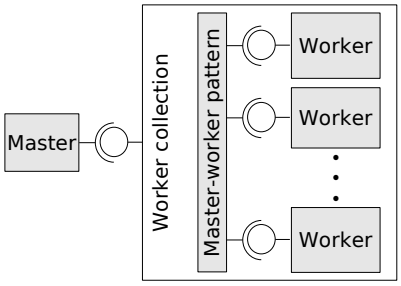


Fig. 3. The master-worker paradigm, in software components

## III. Support for the dynamic adaptation

In order to switch between master-worker implementations at run-time, we use an abstraction of the paradigm, presented in [6], well suited to its adaptation: a component-based collection of workers. The principle of a collection abstracts the paradigm from the distributed systems architecture, while the use of *software components* provides the separation of concerns needed to adapt the collection of workers. A *software component* [7] is a software entity, with a well defined behavior , that can be composed with other components using *ports* (either *sender* or *receiver* ) for communication between them.

In [6], the authors suggest to represent each master and worker in a component. The developers of the final application could write it as if there would be only one master and one worker connected together; where, in fact, worker components are put together in a collection component, as shown on Figure 3. The collection uses a master-worker pattern to handle the communication between masters and workers and the distribution of the requests. A pattern can use any implementation of the paradigm, like NetSolve or DIET.

Despite that the number of workers and the pattern can be chosen when deploying the application, it was not studied how to do it dynamically, when a need or opportunity arise to change the configuration.

In the article [8], the authors study how to make this collection of workers dynamically adaptable and suggest to use the *Dynaco* [9] framework to adapt it. This framework is to be integrated in the component to adapt; here, the collection component. It divides the dynamic adaptation in four major parts: the *monitoring* of constraints, the *decision*, the *planning* and the *execution*. Thus, to use this framework, one has to connect it to probes which gather the information needed to make a decision and to plan the adaptation. This information is gathered passively by an *observer* interface and actively by a *monitor* interface. Then, one has to provide three components to do the last three functions, as shown in the Figure 4. The *decider* component decides when to adapt and sends a *strategy* to the *planner* component when an adaptation is needed. The *planner* component then breaks down the



Authorized licensed use limited to: UR Rennes. Downloaded on December 14, 2009 at 04:42 from IEEE Xplore. Restrictions apply.

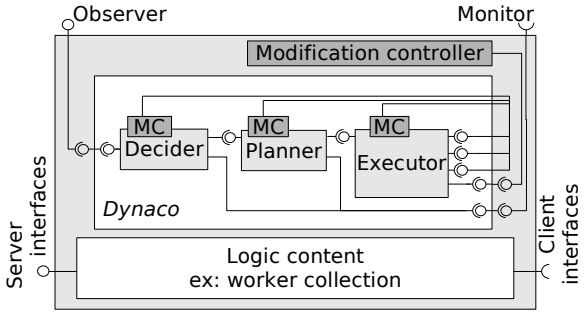


Fig. 4. Dynaco

strategy into an action *plan*, which is a set of elementary tasks. This plan is then sent to the *executor* component which can adapt the collection according to the plan, using *modification controllers*.

In our work, we focused on the *decision* part of the adaptation, which will be presented in Section V But first, Section IV studies what to monitor in order make adaptation choices.

## IV. Characterization of the master-worker paradigm on distributed systems

In this study, we assume that the implementation of the paradigm is limited to one master, one worker collection already deployed and one master-worker pattern. The decision algorithm which has to choose when to switch between master-worker patterns needs to monitor the master-worker application and the distributed system in order to have enough information to make its decisions. To discriminate the pertinent parameters among all of the possible ones, we first have to state *why* to adapt and *what* is to adapt.

The aims of the dynamic adaptation of the master-worker paradigm are manifold. In our study, we focused to those defined from an user point of view, in contrast to a computing resources manager. Besides, since it is impossible to be exhaustive, we studied only the following objectives:

- Maximize the execution speed of the application. It is measured by the average number of executed requests by second. We assume that every pattern ensure that every request sent by the master is processed (i.e. there is no starvation).
- Minimize the processing time of any request, independently of the other requests. This objective should not be confused with the preceding one. This objective enables to consider the case where only some of the requests' processing time are to be minimized.
- Respect a maximum time limit to process a request. This objective enables to design "real-time" applications. It is to the user to specify reasonable time limits, i.e. that can be respected, otherwise the respect of this constraint cannot be ensured.

Other objectives such as respect of a maximum cost or minimizing the processing time of a pack of requests, can be thought of, but they are not studied in this paper.

Now that we know why the application have to be adapted, we have to define what is to adapt. Two parameters of the worker collection can be adapted dynamically.

Firstly, the number of workers can be adjusted as the number of simultaneous requests to be processed evolves with the time, while keeping this number inside the boundaries delimited by the hardware. Also, it can be adjusted when computers appear or leave the distributed system.

Secondly, the master-worker pattern can be changed when evolutions of the distributed system or the type of requests being processed make a unused pattern more suitable to these new conditions.

We focused on three representative patterns in our study: *round-robin*, *load-balancing* and a modified version of *DIET*. Experiments were done with these patterns on the platform *Grid'5000*[1]. Results are given in [10].

A worker is said to be *free* when it can handle a new request sent by the master without affecting the possible other requests being processed by the worker. A worker might only be able to handle one request at the same time.

A function that estimate the extra cost in time due to the pattern must be provided for each pattern, for the decision algorithm. This function is called the *extra cost function*. It can be implemented statically, which can be best suited to simple patterns like round-robin or load-balancing, or by using a learning mechanism, which would be better suited to more complex patterns, like DIET.

*a) Round-robin:* This pattern distributes the number of request equally among the workers.

To this end, the pattern keeps a list of workers in a ring and each worker keeps a queue of requests. Each request sent by the master is transmitted to the worker next to the last who were sent one. Each request received by a worker is put in the queue. The worker process the request in the incoming order and can process one or many requests at a time (for example, one by CPU core).

From the results of the experiments done by Hinde L. Bouziane in her PhD thesis [10], the extra cost function can be approximated by:
$extraCost = a \cdot numberOfWorkers + b$ (in seconds), where $a \approx 9.97 \cdot 10^{-6}$ s and $b \approx 1.62 \cdot 10^{-2}$ s.

*b) Load Balancing:* This pattern distributes the workload equitably among the workers.

It handles a single queue of requests. When a request is received by the master, it is added to the queue. When a request comes to an empty queue, it is stored if there is no free worker, else it is send randomly among the free workers. When a worker send the result of a request, a request from the master's queue is sent back to it if the queue is not empty.

The extra cost function is the same than round-robin's one with $a \approx 1.46 \cdot 10^{-5}$ s and $b \approx 1.62 \cdot 10^{-2}$ s.

[1]Grid'5000: https://www.grid5000.fr (2008)



Authorized licensed use limited to: UR Rennes. Downloaded on December 14, 2009 at 04:42 from IEEE Xplore. Restrictions apply.

*c) DIET:* This pattern uses a request sequencing policy, it uses a distributed architecture with many agents and has probes to its disposal to estimate the workers' processing speed.

We use a modified version of DIET, which differs from the original by a scheduler which sorts the requests by the estimated time to process the requests, when possible.

A single queue of requests is managed by DIET. If an estimation of the time to process each request is available, the queue is sorted in decreasing order of the estimated lengths (we assume there is no starvation or that a priority mechanism is used to avoid it). When a request is received by the master, it is added to the queue. When a request comes to an empty queue it is stored if there is no free worker; else a request is sent to each free worker to estimate and compare their processing speed, then the pending request is sent to the fastest free worker. When a worker sends the result of a request, if the queue is not empty, the master probes the workers to send a request to the fastest one.

Our extra cost function would require further experimental results to refine it and so is not detailed here.

For round-robin, no mechanism is provided to manage requests having to be processed in a limited time. Whereas for the Load balancing and DIET patterns, an estimation of the time to process (in number of cycles) have to be provided by each request if the user chooses to put to each request a time limit to process it. Furthermore, in this case, every request must have a time limit to avoid a starvation, and the queue is then sorted by limit date to send the requests for them to be executed in time.

In order to be able to decide to adapt, the application needs to monitor itself and the distributed system. To this end, we have to select relevant parameters for the adaptation. We consider a potential parameter as useful if its modification can generate a need to adapt, or if it can be used to describe the state of the application's part to adapt (here, the number of workers and the master-worker pattern).

These parameters have to be measurable or known by the application. The list follows:

*Known parameters:* They do not need to be measured. They are composed of: the number of workers, the pattern used and the number of requests being processed.

*Direct parameters:* They have to be measured. They are those like "the time to process a request" or "the workload of each workers" which are needed to compute the *indirect parameters*; and "the number of workers" which can differ from the known parameter, for example in case of a failure.

*Indirect parameters:* They are to be computed (for conciseness we do not describe how do do it):

- The variability of the workload of the workers due to the environment exterior to the workers,
- The variability of the processing power of the workers due to the execution of requests,
- The variability of the time to execute request,
- The time to transmit a request (sending the request and receiving the results) in function of the data's size to transmit, average bandwidth and average delay (from the master to the workers),
- The heterogeneity of the network
- The maximum number of workers, depending on the number of computers.

## V. Decision Algorithm to Change the Pattern

There are at least two ways to adapt the collection. The first is to change the number of workers depending to the workload and the available computing resources. This is not described in this paper for the sake of conciseness. The second way is to dynamically change the master-worker pattern. This paper only deals with this later algorithm.

The decision algorithm is based on the description of the behavior of the patterns, which depends on the state of the pattern and the distributed system, and on a QoS objective.

It is designed to be generic: new patterns can be added to the application without modifying the existing parts of the algorithm implementation. In addition, the algorithm can be adapted once the application is deployed, thanks to Dynaco framework's dynamic adaptability. To this end, the behavior description of a pattern is independent from those of the other patterns.

It is also designed to be added to a learning mechanism which could tune its decisions according to the results of preceding adaptations. However, such mechanism is not compulsory.

The algorithm is a compromise between performance and an ideal solution. Indeed, It does not aim to select the pattern the best fitted to every situation. Instead, it aims to discriminate a pattern among the best fitted in a short time. Moreover, it is not always useful to select the optimal pattern among two almost equivalent ones, as long as the inadequate ones are filtered.

The principle of the algorithm is to compare the patterns using a description of their behavior. This comparison is done using positives scores, including $+\infty$: the pattern with the lowest score is the best fitted to the situation for the given QoS.

The behavior of a pattern don't have to be set for each QoS objectives, but a pattern can only be used when its behavior is set for the selected QoS objective.

The behaviors of the patterns are divided into elementary behaviors for which a *cost function* is defined. Each of these functions is based on a *characteristic* which is built from parameters taken among those presented previously.

The aim is to provide functions representing approximatively the extra cost due to each characteristic. In this sense, there is no need to find the exact functions, knowing that the weighting refines them. Furthermore, since the user might want to use its application to compute very short tasks, it is important for the computation of the indices to be fast, which limits the possible complexity of the functions.

These functions can be discovered using simulations, by monitoring the behavior of the pattern in controlled environment or by knowing how the pattern behaves.



Authorized licensed use limited to: UR Rennes. Downloaded on December 14, 2009 at 04:42 from IEEE Xplore. Restrictions apply.

We have identified nine characteristics such as the number of workers, the number of requests being processed, the heterogeneousness of the network.

For each of these characteristics an index is computed by the cost function, that represents the extra cost resulting of the characteristic.

Once the indices are computed, they are weighted and added to make a score by pattern. Then the scores are compared; the pattern with the lowest score is selected. If the difference between this score and the score of the pattern currently used is greater than $5\%$ of the latter (that is, if the difference between the scores is significant), then an adaptation is triggered. If all the scores are infinites, there is no lowest score, so there is no pattern selected and no adaptation triggered. In case of conflict between patterns, one is to be arbitrary selected, the last used might be a good choice to avoid the overhead of deploying a new pattern.

We studied the behavior of the three patterns for the three QoS objectives.

### A. Simulations

To study the dynamic adaptation of the master-worker paradigm, we made simulations and used experimental results from work done in our project-team, published in Hinde L. Bouziane's PhD thesis [10].

To validate the decision algorithm, two engines were developed: one to simulate master-worker patterns on distributed systems, the other to take decisions according to the algorithm. The first one is used to simulate the behavior of the various schedulers of the patterns and characteristics of the distributed systems (like the computers workload). The second one offers to simulate the evolution in the time of the parameters used in the decision algorithm, in order to visualize adaptation choices done by the algorithm.

To give a rough idea of the size of these simulators: they are written in Java and range from 1200 to 1500 lines of code. For its decision engine, the simulator of adaptation choices uses the organization of the *decision* part of Dynaco (c.f. Section III).

The first simulator is used to study the behavior of the patterns by varying the workers workload, the time to process requests, the measurement and prediction errors of computation time, in function of various distributions of random numbers.

On top of the three presented patterns (one with and one without probes for DIET), an optimal pattern, without extra cost and without measurement and prediction errors, is implemented. It is used to calibrate the indices, such that, as written before, an index of $1$ means a performance gap of $5\%$ with an optimal pattern; the performance measure being relative to the QoS objective.

For each objective, simulations can be done by varying the value of one characteristic at a time. It enables to verify that the patterns behave as expected and to ponderate the characteristics by pattern or, if needed, to correct the behavioral functions.

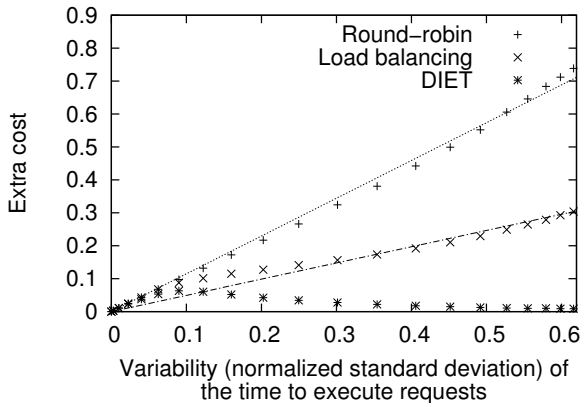


Fig. 5. Extra cost rate w.r.t. an optimal distribution of requests, due to the variability of the time to execute requests

For example, Figure 5 shows the extra cost due to the *variability of the time to execute requests* when the objective is to maximize the application's execution speed. It shows clearly that this extra cost can be approximated linearly for round-robin, by a function of equation $y = 1.15 \cdot x$. We can also notice that the extra cost tends to be linear for the load-balancing pattern ($y = 0.49 \cdot x$), except for small values where it is not efficient. For DIET, we notice that the prediction of the time to execute requests (here with an average prediction error of $8\%$) becomes efficient when the variability increase. This shows that we can use the function $y = 1.15/0.05 \cdot x$ for round-robin, $y = 0.49/0.05 \cdot x$ for load-balancing and $y = 0$ for DIET.

This simulator has been used to validate, refine or correct half of the nine characteristics. For the others, either they do not need this work since their behavior is well known, as for example "number of requests being processed" ; or the simulator describes not precisely enough the exact behavior of the pattern to be able to draw conclusions from it, which is the case for the pattern DIET and the characteristic "the number of requests being processed divided by the number of workers". This limitation can be lifted by doing complementary measurements of those used in Section IV

The second simulator is used to simulate which adaptation choices are done and when they are done, when the characteristics of the distributed system and the master-worker collection evolve with time. Thus it can ensure the consistency of the choices and their relevance. This simulator is to be used in concurrence with the first one to check if the choices done were the choices to be done. It can be used in a real test phase to plan interesting test scenarios.

For example, it was used to validate the choice to require to use a score gap of $5\%$ between the current pattern and the replacing one before to exchange patterns. It was also used to compute the indices presented in Section V



Authorized licensed use limited to: UR Rennes. Downloaded on December 14, 2009 at 04:42 from IEEE Xplore. Restrictions apply.

## VI. Conclusion

Today, it is still difficult to write applications for distributed systems. Thus, we chose to use the master-worker paradigm to get a high level abstraction of the system and to implement it in software components in order to adapt it. Then, we presented a framework to adapt the component-based master-worker collection. We then focused on an algorithm to choose when to change the master-worker pattern. To this end, we first characterized the distributed system and the master-worker collection, then we used this characterization to build the decision algorithm, then we briefly presented the simulations we used to validate it.

Through this paper, we described the process of designing the dynamic adaptation of the master-worker paradigm. Using the experimental plateform remains to be done in order to completely validate our algorithm. Then, it would be possible to use this design to build a framework for parametric and distributed applications which take into account the dynamic behavior of these applications and their execution environment.

In this study, we focused on the conception of algorithms to decide when to adapt and which pattern to choose. However, for the final conception of the global framework, it would be interesting to better study how to switch between patterns efficiently. It would also be appropriate to study the use of several masters or several worker collections inside the same application. In addition, it would be relevant to study ways to specify the QoS objectives more formally, to enable the user to have a better control over the dynamic adaptation.

## Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube) and from the French National Agency for Research project LEGO (ANR-05-CIGC-11).

## References

[1] D. Anderson, S. Bowyer, J. Cobb, W. S. D. Gbye, and D. Werthimer, "A new major SETI project based on Project SERENDIP data and 100,000 personal computers," in *Astronomical and Biochemical Origins and the Search for Life in the Universe*. Bologne, Italie: IAU Colloquium 161, 1997, p. 729.

[2] *Grid Computing and New Frontiers of High Performance Processing*. L. Grandinetti, Elsevier, 2005, ch. NetSolve: Grid Enabling Scientific Computing Environments.

[3] P. Combes, F. Lombard, M. Quinson, and F. Suter, "A Scalable Approach to Network Enabled Servers," in *Proceedings of the 7th Asian Computing Science Conference*, janvier 2002, pp. 110–124.

[4] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing," in *J. Grid Computing*, vol. 1, no. 1, 2003, pp. 41–51.

[5] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An Architecture for a Ressource Management and Scheduling System in a Global Computational Grid," in *High-Performance Computing*, vol. 1, no. 1. Chine: IEEE CS Press, USA, 2000, p. 283.

[6] H. Bouziane, C. Pérez, and T. Priol, "Modeling and executing Master-Worker appplications in component models," in *11th Int. Workshop on HIPS*, Rhodes Island, Greece, avril 2006.

[7] C. Szyperski, *Component Software: Beyond Object Oriented programming*. Addison-Wesley Longman Publishing Co., Inc, 2002.

[8] F. André, H. L. Bouziane, J. Buisson, J.-L. Pazat, and C. Pérez, "Towards Dynamic Adaptibility Support for the Master-Worker Paradigm in Component Based Applications," in *Corregrid Symposium*, Rennes, France, août 2007.

[9] J. Buisson, F. André, and J.-L. Pazat, "Supporting adaptable applications in grid resource management systems," in *8th IEEE/ACM International Conference on Grid Computing*, Chicago, USA, 19-21 September 2007.

[10] H. L. Bouziane, "De l'abstraction des modèles de composants logiciels pour la programmation d'applications scientifiques distribuées," Ph.D. dissertation, École Doctorale MATISSE, février 2008.



Authorized licensed use limited to: UR Rennes. Downloaded on December 14, 2009 at 04:42 from IEEE Xplore. Restrictions apply.